Software and the Concurrency Revolution

Herb Sutter

Software Architect
Microsoft Developer Division

The Last Slide First

What you need to know about concurrency

It's here

parallelism has long been the "next big thing" – the future is now

everybody's doing it (because they have to)

It will directly affect the way we write software

the free lunch is over – for sequential CPU-bound apps

only apps with lots of latent concurrency regain the perf. free lunch

(side benefit: responsiveness, the other reason to want async code)

languages won't be able to ignore this and stay relevant

The software industry has a lot of work to do,

and we suspect the HW industry vastly underestimates that

a generational advance >OO to move beyond "threads+locks"

key: incrementally adoptable extensions for existing languages

Concurrency

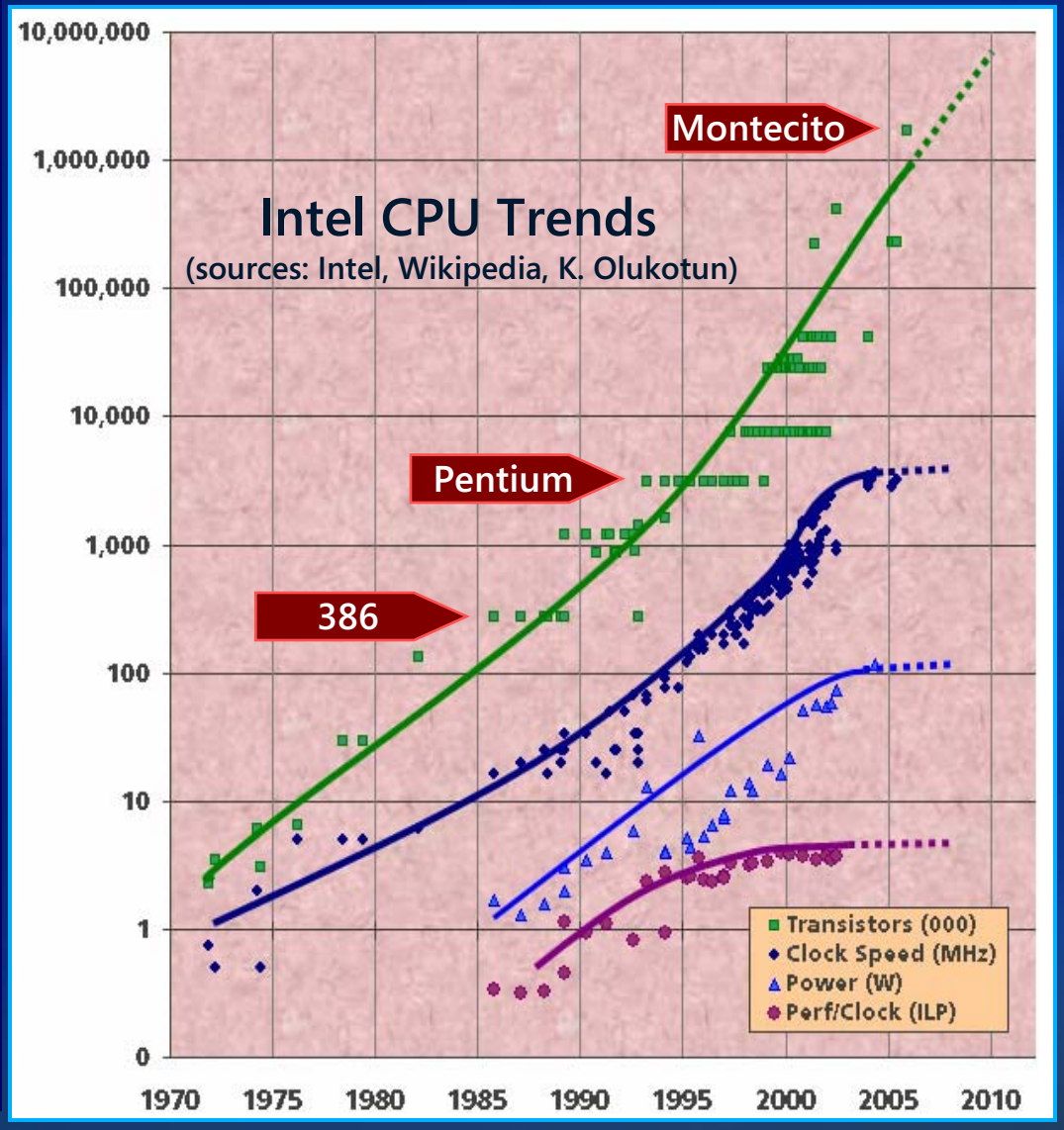
Truths

Consequences

Futures

The Free Lunch Is Over

Unless we reenable it…



- Nonconcurrent apps aren't going to get faster (expect some regressions). We need killer apps with lots of latent parallelism.
- The state of concurrency in the software industry is terrible. Nobody does concurrency well for mainstream languages and platforms. The world's best frameworks are just getting away with it.
- A generational advance >OO is necessary to get above the "threads+locks" programming model.

The Issue Is (Mostly) On the Client

What's "already solved" and what's not

"Solved": Server apps (e.g., database servers, web services)

lots of independent requests – one thread per request is easy

typical to execute many copies of the same code

shared data usually via structured databases

(automatic implicit concurrency control via transactions)

⇒ with some care, "concurrency problem is already solved" here

Not solved: Typical client apps

somehow employ many threads per user "request"

highly atypical to execute many copies of the same code

shared data in memory, unstructured and promiscuous

(error prone explicit locking – where are the transactions?)

also: legacy requirements to run on a given thread (e.g., GUI)

Dealing With Ambiguity

	Sequential Programs	Concurrent Programs
Behavior	Deterministic	Nondeterministic
Memory	Stable	In flux (unless private, read-only, or protected by lock)
Locks	Unnecessary	Essential
Invariants	Must hold only on method entry/exit, or calls to external code	Must hold anytime the protecting lock is not held
Deadlock	Impossible	Possible anytime there are multiple unordered locks
Testing	Code coverage finds most bugs, stress testing proves quality	Code coverage insufficient, races cause hard bugs, and stress testing gives only probabilistic comfort
Debugging	Trace execution leading to failure; finding a fix is generally assured	Postulate a race and inspect code; root causes easily remain unidentified (hard to reproduce, hard to go back in time)

A Final Word on "Truths"

Don't underestimate the programming problem.

The hardware community is building parallel hardware, but do you recognize how hard it is to program?

Don't assume the guy upstream can and will solve the hard problems.

This talk will show ideas on future software directions, but these aren't (yet) proven solutions or shipping products.

Hardware semantics and operations should focus on programmability first, speed second.

In particular, non-sequentially consistent memory models are an enormous source of difficulty for programmers.

See for example "Multiprocessors Should Support Simple Memory Consistency Models," Mark D. Hill, IEEE Computer, August 1998. Affirmed at Dagstuhl 2003.

Software can help mitigate: Try to keep both SC and performance by reducing/eliminating mutable shared state. (Easy to say…)

Concurrency

Truths

Consequences

Futures

A Software Revolution

Motivating an "OO for concurrency"

Concurrency is likely to be more disruptive than OO

Languages can't ignore it

languages could ignore OO and remain relevant (e.g., C)

today's languages will be forced to add direct support for concurrency, or be marginalized to non-demanding apps

It's demonstrably harder

e.g., analysis that is routine for sequential programs is provably undecidable for concurrent programs

We need higher-level abstractions for mainstream languages

"threads + locks" ≡ structured programming

necessary new abstractions ≡ objects

Today's Status Quo Isn't Enough
The good, the bad, and the ugly

Problem 1: Free threading

e.g., arbitrary affinity, blocking, reentrancy

willy-nilly concurrency yields higgedly-piggedly failures

explicit threading is too low-level

Problem 2: Mutable shared memory + locks

locks are the best we have, **but aren't composable**

locks are hard for expert programmers to get right

("lock-free" isn't an answer; that's hard for geniuses to get right)

All current mainstream languages' concurrency support

based on threads + locks

Toward an "OO for Concurrency"

What we need for a great leap forward

What: Enable apps with lots of latent concurrency at every level

cover both coarse- and fine-grained concurrency,
from web services to in-process tasks to loop/data parallel

map to hardware at run time ("rightsize me")

How: Abstractions (no explicit threading, no casual data sharing)

active objects asynchronous messages futures
rendezvous + collaboration parallel loops

How, part 2: Tools

testing (proving quality, static analysis, …)
debugging (going back in time, causality, message reorder, …)
profiling (finding convoys, blocking paths, …)

O(1), O(K), or O(N) Concurrency?

1. Sequential apps.

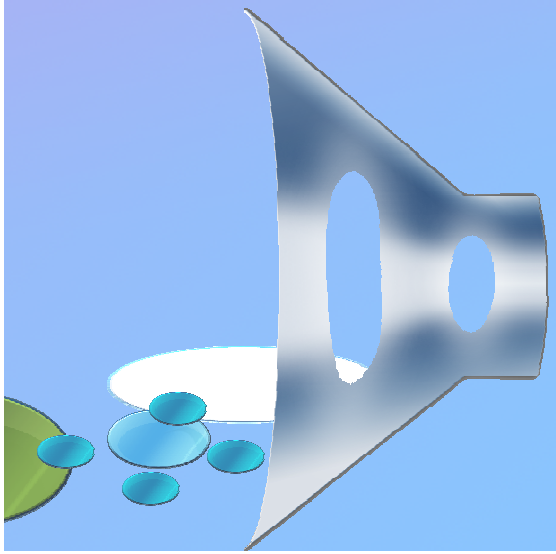
- The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
- Potentially poor responsiveness.

2. Explicitly threaded apps.

- Hardwired # of threads that prefer K CPUs (for a given input workload).
- Can penalize <K CPUs, doesn't scale >K CPUs.

3. Scalable concurrent apps.

- **Workload decomposed into a "sea" of heterogeneous work items (with ordering edges).**
- **Lots of latent concurrency we can map down to N cores.**



O(1), O(K), or O(N) Concurrency?

The bulk of today's client apps

1. Sequential apps.

- The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
- Potentially poor responsiveness.

Virtually all the rest of today's client apps

2. Explicitly threaded apps.

- Hardwired # of threads that prefer K CPUs (for a given input workload).
- Can penalize <K CPUs, doesn't scale >K CPUs.

Essentially none of today's client apps

(outside limited niche uses, e.g.: OpenMP, background workers, pure functional languages)

3. Scalable concurrent apps.

- **Workload decomposed into a "sea" of heterogeneous work items (with ordering edges).**
- **Lots of latent concurrency we can map down to N cores.**

Concurrency

Truths

Consequences

Futures

Concurrency Tools in 2005 and Beyond

Concurrency-related features in recent products:

- OpenMP for loop/data parallel operations (Microsoft, Intel).
- Memory models for concurrency (Java, .NET, C++).
- New/experimental work: Fortress (Sun), Cω (Microsoft), transactional memory research (Intel, Microsoft, Sun, …).

The rest of this talk is about futures (pun intended):

- The Concur project goals & details.
- NB: There's lots of other work going on at MS.
  This happens to be mine.

Concur Goals

The Concur project aims to:

- define higher-level abstractions
- for today's imperative languages
- that evenly support the range of concurrency granularities
- to let developers write correct and efficient concurrent apps
- with lots of latent parallelism (and not lots of latent bugs)
- mapped to the user's hardware to **reenable the free lunch.**

Eliminate/reduce "threads+locks":

- **Blocking and reentrancy:** Never silently or by default, always explicit and controlled by higher-level abstractions.
- **Isolation:** On active object boundaries + ownership semantics (e.g., transfer/lending). Reduce mutable sharing & locking.
- **Locks:** Declarative support for associating data with locks, expressing lock levels, etc. Support static/dynamic analysis.

50,000' View: Producing the Sea

1. Active objects with async method calls.

```
active C c;
c.f();                  // these calls are nonblocking; each method
c.g();                  // call automatically enqueues message for c
...                     // this code can execute in parallel with f & g
```

2. Async calls/work via active closures.

```
x = active { /*...*/ return foo(10); }; // do some work asynchronously
y = active { a->b( c ) };               // evaluate asynchronously
z = x.wait() * y.wait();                // express join points via futures
```

3. Parallel loops.

```
for each( Employee e in emps ) active { ... } // parallel loops
for( i = v.begin(); i != v.end(); ++i ) active { // composed of
  ...                                            // work items
}
```

Gaining/losing concurrency is explicit: active and wait.

Active Objects and Messages

Nutshell summary:

- Each active object conceptually runs on its own thread.
- Method calls from other threads are async messages processed serially ⇒ atomic w.r.t. each other, so no need to lock the object internally or externally. Default mainline is a prioritized FIFO pump.
- Return values and out parameters are futures (future<T>).
- Expressing thread/task lifetimes as object lifetimes lets us exploit existing rich language semantics.

```
active class C {
public:
  void f() { … }
};
```

```
// in calling code, using a C object
active C c;
c.f();                  // call is nonblocking
…                       // this code can execute in parallel with c.f()
```

Futures

Return values are future values:

- Return values (and "out" arguments) from async calls cannot be used until an explicit wait for the future to materialize.

```
// in calling code, using a Calc object
future<double> tot = calc.TotalOrders(); // call is nonblocking
... potentially lots of work ...                // parallel work
DoSomethingWith( tot.wait() );                  // explicitly wait to accept
```

Why require explicit wait? Four compelling reasons:

- No silent loss of concurrency (e.g., early "logFile << tot;").
- Explicit block point for writing into lent objects ("out" args).
- Explicit point for emitting exceptions.
- Need to be able to pass futures onward to other code (e.g., DoSomethingWith(tot) ≠ DoSomethingWith(tot.wait())).

Using Futures and Active Lambdas

Active blocks (lambdas) for queueing up work items:

```
x = active { foo(10) };          // call foo asynchronously
y = active { a->b( c ) };        // evaluate asynchronously
p = active { new T };            // allocate and construct asynchronously
... more code, runs concurrently with all three active lambdas ...
return x.wait() * y.wait() * p.wait()->bar();
```

Idioms:

- "Active" to call a synchronous function asynchronously:

```
active { plainObj.Foo(42) };          // type is future<ReturnType>
```

- "Wait" to call an async function synchronously:

```
    activeObj.Bar(3.14).wait();       // type is ReturnType
or  wait( activeObj.Bar(3.14) );
```

- "Active…wait" to get outside locks and leave caller interruptible:

```
active { SomeLongOperation() }.wait();
```

Comparison: Async Pattern + Delegates

```
p

  public static v
    int cookieVa
    // ... here, ch
    int result = (
      ((AsyncResu
    Console.Writ
  }
}
```

```
int main() {
  future<int> result = active { sampSyncObj.Square( 42 ) };
  // ... do useful work ...
  Console.WriteLine("Result is {0}", result.wait());
}
```

OpenMP and Beyond

OpenMP offers many benefits:

- Portable, scalable, flexible, standardized, and performance-oriented interface for parallelizing code.
- Hides many details: Thread team created at app startup, per-thread data allocated when #pragma entered, and work divided into coherent chunks.

But it also has many limitations:

- C and Fortran – not C++, C#, Pascal, Python, VB, etc.
- Outside the language, bolted onto code via #pragmas.
- Only good for simple *for* loops over arrays (doesn't work with STL or .NET collections), and for parallel code sections.
- Doesn't take advantage of modern languages' superior abstractions for type safety or generic programming.

Example: Average-Neighbors

OpenMP code (example a1):

```
#pragma omp parallel for
for( i=1; i < n; ++i )
  b[i] = (a[i] + a[i-1]) / 2.0;
```

Concur code (2 options):

- Any kind of loop can have an active body:

```
for( i=1; i < n; ++i ) active {
  b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- A higher-level loop:

```
for each( int i in range(1,n) ) active {
  b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- Both work also with STL and .NET collections, not just arrays.

Example: Parallel Array Update

OpenMP code:

```
void a7( float *x, int *y, int n ) {
  float a = 0.0;
  int b = 0, i;
  #pragma omp parallel for reduction(+:a) reduction(^:b)
  for( i=0; i<n; i++ ) {
    a += x[i];
    b ^= y[i];
  }
}
```

Concur code:

```
void a7( vector<float> &x, vector<int> &y ) {
  float a = paccumulate( x, 0.0, _1 + _2 );
  int   b = paccumulate( y, 0,   _1 ^ _2 );
}
```

- Also, may get better cache performance by going through one collection and then the other. (The original code may have bad cache behavior unless x and y are related in some way.)

Concurrency

Truths

Consequences

Futures

The First Slide Last

What you need to know about concurrency

It's here

parallelism has long been the "next big thing" – the future is now

everybody's doing it (because they have to)

It will directly affect the way we write software

the free lunch is over – for sequential CPU-bound apps

only apps with lots of latent concurrency regain the perf. free lunch

(side benefit: responsiveness, the other reason to want async code)

languages won't be able to ignore this and stay relevant

The software industry has a lot of work to do,

and we suspect the HW industry vastly underestimates that

a generational advance >OO to move beyond "threads+locks"

key: incrementally adoptable extensions for existing languages

Questions?